Cooperative Update: A New Model for Dependable Live Update

Cristiano Giuffrida
Department of Computer Science
Vrije Universiteit, Amsterdam
c.giuffrida@few.vu.nl

Andrew S. Tanenbaum
Department of Computer Science
Vrije Universiteit, Amsterdam
ast@cs.vu.nl

Abstract

Many real-world systems require continuous operation. Downtime is ill-affordable and scheduling maintenance for regular software updates is a tremendous challenge for system administrators. For this reason, live update is a potential solution as it allows running software to be replaced by a newer version without stopping the system. The vast majority of live update approaches proposed as a solution to this problem aims to support existing software systems, while striving to maintain a good level of safety and flexibility.

In this paper, we consider the opposite direction. Our work aims to build dependable and trustworthy live updatable systems that do not attempt to be backward compatible but look forward to solving the update problem in future systems. To this end, we highlight possible issues and limitations in existing approaches and propose a new *cooperative* model for live update to provide better safety and flexibility guarantees.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; C.4 [*Performance of Systems*]: Reliability, Availability, and Serviceability

General Terms Design, Maintenance, Dependability

Keywords DSU, Live Update, Update Validity

1. Introduction

Experience indicates that trading off high availability against the need to update a software system is painful for many systems. Live update—namely the ability to update software without service interruption—is a promising direction to address this problem. An infrastructure to apply online changes to a running system would greatly aid in the maintenance of systems that cannot tolerate disruption of service or loss of transient state. Live update could be the definitive solution to support software evolution in high-availability environments. Unfortunately, despite significant effort, few research systems for software-based live update have made their way into the real world. The majority of high-availability systems still rely on hardware-based solutions or do not use live update at all.

In this paper, we investigate the root causes of the poor acceptance, review existing solutions, and highlight possible issues and limitations. We argue that existing models do not scale efficiently for complex systems and nontrivial general-purpose software updates. We then present a new *cooperative* model for live update. In our approach, the old and new versions actively cooperate in the live update process by being able to understand the nature of the update and react accordingly. Our model could be applied at different levels of granularity. In the paper, we assume a generic component as the structural unit of dynamic replacement. The model aims to achieve better dependability properties at the cost of no backward compatibility with legacy software already out in the field.

The remainder of the paper is as follows. We first discuss existing models for live update highlighting characteristics, limitations, and trade-offs (Sec. 2). Then we present examples of updates of different natures, analyze their implications, and present our model (Sec. 3). Section 4 finally presents our conclusions.

2. Existing models

System research in the area of live update is generally focused on designing frameworks to seamlessly apply online changes to existing software systems the instant they arrive.

We observe three different and separate worlds: the world of software developers, the world of update authors, and the world of system administrators. Developers build the software unaware of live update. Update authors fetch an existing patch or two different versions of the system and build a live update patch with the help of the live update infrastructure. A live update patch typically contains code from the new version and an author-provided state transfer function to transform the state of the system at update time into an

HotSWUp'09, October 25, 2009, Orlando, Florida, USA.

equivalent state for the new version. The patch is finally delivered to system administrators, who are instructed to install it on-the-fly the instant the update is made available. Figure 1 depicts the evolution of a software system from version $V1$ to version Vn by installing a series of live update patches LU.

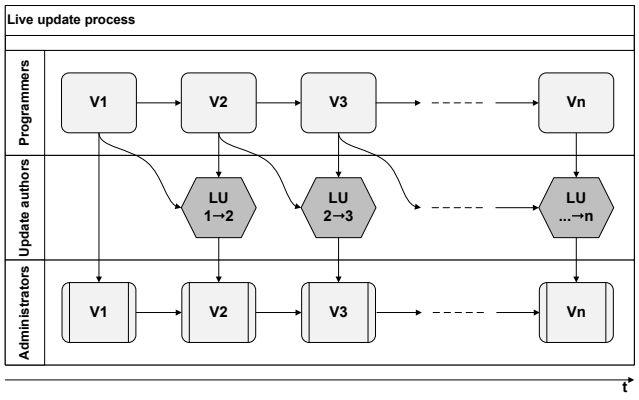

Figure 1. The common model adopted for live update.

This model has largely promoted performance and transparency as key success criteria for live update technologies. The extensive focus on these properties has, however, led many researchers to neglect other important characteristics. We now analyze these properties in more detail and justify our claim further.

Is performance necessary?
Performance measurements have been extensively used to determine the time and the overhead required to apply an online change. Much recent research has specifically focused on improving update timing to maximize performance, even at the cost of additional complexity [1, 2]. In practice, fast update is rarely a concern in real-world scenarios.

Consider bug fixes. It usually takes a very long time to find a bug, fix it, and publish a patch. Studies on operating systems have estimated a mean bug lifetime of about 1.8 years, with the median around 1.25 years [3]. Even once a bug is discovered, it may take a considerable amount of time to resolve it and deliver an adequate patch. A recent study shows that many difficult (but possibly critical) bug reports may be deferred for months or, in some case, indefinitely [4].

Consider security patches. Although patching soon is desirable not to expose the system to possible attacks, experience indicates that timing the installation of updates is crucial to avoid problems induced by possible bugs in the patch itself. Previous work on security patches suggested that system administrators should delay an update at least 10 days after the patch's release [5].

In light of all these observations, delaying a live update for a few seconds in order to get the system into a simpler and known state to make updating easier and less error prone seems to be a reasonable option.

Is transparency desirable?
Transparency relates to the property of hiding the nature of the update and details of the live update process from software developers, update authors, system administrators, and the system itself. In most studies this property is regarded as the ability of the framework to support existing binary or source patches and dynamically apply them starting from an arbitrary state of the system with no or little modification.

The focus on transparency in prior work is largely motivated by the need to cope with existing systems. While its importance in supporting legacy systems is indisputable, this model has many known practical limitations derived from its inherent implementation complexity. Supporting a broad range of updates efficiently while ensuring that the resulting configuration is valid is hard in the general case [6].

```
1  void task(int t) {
2    init(t);
3    run(t);
4  }
5  void init(int t) {
6    count++;
7    do_init(t);
8  }
9  void run(int t) {
10   do_run(t);
11 }
```

(a) Version 1

```
1  void task(int t) {
2    init(t);
3    run(t);
4  }
5  void init(int t) {
6    do_init(t);
7  }
8  void run(int t) {
9    count++;
10   do_run(t);
11 }
```

(b) Version 2

Figure 2. Two different versions of a program fragment in a C-like syntax.

Figure 2 presents an example of an update scenario for a program in a C-like syntax. Assume the program fragment in Figure 2a must be dynamically replaced by the fragment in Figure 2b. Many existing live update solutions cannot deal with this simple scenario efficiently. Approaches focusing exclusively on type safety would allow the execution of the new version of $run()$ right after the execution of the old version of $init()$ [1, 7–11]. In that case, the update would bring the system to an invalid state with the variable $count$ incremented twice. Note that no simple state transfer function could solve this problem, under the assumption that the update is performed in an arbitrary state of the system.

A common solution to this problem is to limit the set of live updates allowed. Update authors are expected to laboriously manually inspect the code of the two versions of the system to figure out whether the resulting live update can be reliably performed starting from an arbitrary state of the system. The main problem with this approach is that it does not scale because two versions may have thousands of changes. While it is easy to understand that the example above would not produce correct results in the general case, the same analysis on more complicated updates can become expensive and error prone, possibly leading to an unreliable live update pro-

cess. To make manual inspection realistic, a number of approaches, such as OPUS [7] and Ksplice [11], specifically target small security patches. For example, Ksplice has been evaluated on a list of significant Linux CVE (Common Vulnerabilities and Exposures) patches from May 2005 to May 2008 [12]. Figure 3 shows that the median number of lines of code changed or added in these patches is only 4, while the median is close to 100 for standard Linux incremental patches (e.g. from 2.6.11.1 to 2.6.11.2) of the same period. Further, if we consider minor version change patches (e.g. from 2.6.11 to 2.6.12), statistics like the ones presented in Figure 3 reveal results on the order of hundreds of thousands of lines of code. In our work, reliability and scalability are major concerns, as we aim to support efficiently many types of updates, ranging from small bug fixes or security patches to new versions of the system with additional functionalities.

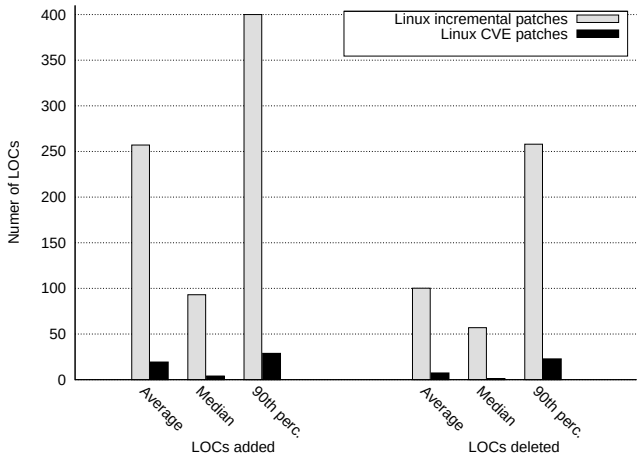

Figure 3. Comparative LOC analysis of Linux incremental patches and Linux CVE patches from May '05 to May '08.

For multithreaded applications, manual inspection can become even more complicated because the number of possible states of the system grows exponentially with the number of running threads. In some cases, manual inspection may also be required to detect possible deadlocks. For example, in [13], the use of manual checking—or a timeout-based update process as a fallback mechanism—is advocated to avoid possible deadlocks when cyclic external call chains occur.

To overcome the issues presented above, a number of approaches go beyond type safety and offer support for additional safety constraints. To make sure that the update is not performed while executing a specific code region, update authors can specify safe update points [14] or mark blocks of code that must be entirely executed on a single version of the system [15]. While these approaches have been effective in some real-world scenarios, they potentially suffer from a structural problem. Update safety constraints are hard-coded in the original version of the system and cannot be modified at update time.

Recall the example in Figure 2. By looking at the structure of the original code, one might conclude that *line 4* in Figure 2a is a safe update point to replace functions $init()$ and $run()$. In practice, assume a new update is published to measure the time it takes to execute all the tasks. If $init()$ is modified to record the start time of the first task and $run()$ is modified to record the end time of the last task, no state transfer function could bring the system to a valid state if the update is executed at *line 4* before all the tasks complete. While this is an artificial example, one might expect to deal with similar issues far more frequently with complex systems and nontrivial updates. At the very least, meticulous manual patch inspection is still required to ensure a reliable update process. Similar problems in real-world scenarios have been reported in [16], while experimenting live updates with *sshd* and *vsftpd*.

Other approaches specific to event-driven systems have suggested using atomicity at the level of an event to make sure that each event-generated transaction is entirely executed on a single version of the system [17]. Supporting atomicity at the level of an event, however, reduces the degree of flexibility and may cause excessive system disruption for small updates [18]. In addition, event-level atomicity may not be sufficient in some cases. It is easy to show that it would not even suffice for the example in Figure 2 and the time measurement update if the execution of the tasks were distributed across multiple events.

As evident from the examples presented, there is a clear tension between the level of transparency ensured and the safety and flexibility properties of the resulting live update process. A key problem with the transparency model is that the nature of any future update is ignored in the original design, thus resulting in very high implementation complexity to provide even basic safety and flexibility guarantees. Another problem relates to the way safety constraints are enforced by the live update infrastructure. If safety constraints are enforced eagerly, namely before the update can take place, the update process may not complete in bounded time. In contrast, if safety constraints are enforced lazily, providing hard guarantees on the correctness of execution becomes even more challenging in the general case.

Assuming we do not aim at transparency and backward compatibility, what kind of system support do we need to maximize safety and flexibility? What level of atomicity is desirable to best achieve these properties? In the following sections, we address these questions in detail.

3. Our Model

Hicks [19] describes two main models of updating, the *interrupt model*—the system is interrupted at an arbitrary point during its execution, the update is performed, and execution is resumed—and the *invoke model*—the system is modified to invoke an update procedure periodically that will result in a live update if a new version is available. Here we introduce

a new *cooperative model*, where the system is notified that an update with certain properties is available, reacts preparing itself, and, only when ready, installs the update online. In our approach the *nature* of the update is central.

The Nature of an Update

In complex systems, different updates may require very different conditions to be applied. When considering several categories of updates, the notion of *transactional version consistency* [15] can be generalized throughout the entire lifetime of a software system. In the following, we draw examples from operating systems to examine a number of update scenarios with different levels of impact. Our analysis does not aim at generality and completeness, but is intended to illustrate and examine a number of concrete examples of updates of fairly different natures.

1. Update affects a single component. This scenario comprises changes isolated in a single component. Common updates at this level are small bug fixes and security patches. An example is a change to the data type of an access counter of a component to avoid data loss (e.g. from an *int* to an *unsigned int*).

2. Update affects interaction between components. This scenario comprises changes to an interaction between two or more components. An example is changing the interface of a call to the disk driver to represent a block number in 48 bits instead of 32.

3. Update affects global data. This scenario comprises changes to global data structures that are shared across multiple components. An example is a change to global data constants, like renumbering all the error codes.

4. Update affects global algorithms. This scenario comprises changes to global algorithms that may affect multiple components. An example is an improved implementation of a file usage counter. Assume the original version incremented a counter in the inode at $open()$ time, while in the new version the counter is incremented when the first $read()$ or $write()$ is processed. Another example is a change to the generation algorithm of the random number generator.

5. Update affects data on the disk. This scenario is generally concerned with data stored on the disk. An example is a change to the format of the disk image used for process checkpointing. Another example is a change to the filesystem format to support the creation time of a file.

6. Update affects hardware requirements. This scenario comprises changes that impose new hardware requirements. An example is the transition from Windows XP to Windows Vista. Minimum requirements went from 64MB to 512MB for RAM and from 1.5GB to 15GB for disk space available.

From the examples above, we observe very different scenarios as we vary the nature of the update. In some cases, the update is not feasible on-the-fly. Consider the random number generator example. If running applications or components of the operating system rely on a sequence of random numbers provided by the generator, a live update would break this assumption regardless of when changes are applied. The only reliable solution here is a conventional reboot update. In other cases, such as the file creation time example or a change imposing new hardware requirements, an update may not be possible at all.

In addition, live updates of different natures may require different levels of atomicity of execution to be applied. In the access counter example, the update could occur immediately, under the assumption that the component code is not executing. In the file usage counter example, we can reliably perform the update only when every open file has already executed at least one $read$ or $write$ operation. In some cases, the level of atomicity required at update time can be relaxed. In the file usage counter example, we could improve update timing if an author-provided state transfer function is able to adjust the value of the counter correctly for each open file.

Nevertheless, as we gradually relax constraints imposed at update time, we observe an increasingly complicated state transfer. In some circumstances, constraints cannot be further relaxed or state transfer will become infeasible. These considerations further suggest that assuming an arbitrary or a fixed state of the system at update time is not desirable. Gupta has formally proven that the validity of a live update is undecidable for an arbitrary state of the system and arbitrary program versions [6]. Although the proof is valid, we reject both of the assumptions it is based on—(i) the update must be applied instantly and (ii) the system is not aware that updates happen. We advocate (i) allowing the system to delay an update for a few seconds in order to get into a known and stable state and (ii) having the system be aware that an update may happen and be prepared to handle it.

The Live Update Process

In the previous section, we discussed how the nature of the update determines the characteristics of the resulting live update process. In our model, we make this explicit. Each update carries with it adequate information to describe what changed and when it can be applied. To realize this vision, we believe a paradigm shift is necessary, moving from the common belief that live update is somehow similar in spirit to conventional patch installations. Our model integrates live update as part of the software development process.

In particular, the software developers producing the update should document the changes they made in a *live update package*. Note that we use the new terminology to indicate the distinction between patch installation and live update. A

live update package contains code to update a running instance of the system to the new version, but also developer-provided metadata to describe the nature of the changes and specifications on how to apply them online properly. The enclosed metadata shall provide semantic information necessary to shape the properties of the update process and drastically reduce the amount of non-determinism one has to deal with at update time.

The system, in turn, should support an infrastructure to interpret developers' specifications and apply the changes correctly at the appropriate time. At the heart of the system lies the update manager, which translates the specifications contained in the live update package into an update protocol. The protocol is used to drive the system into the required state in a cooperative fashion and apply changes online after then. In the following, we discuss the evolution of the live update process in more detail.

1. Initialization. A new live update package is submitted to the update manager, which loads and processes the enclosed information. The package includes the new code for a number of system components and the associated state transfer and state checking functions (if any). The metadata in the package specify the set of affected components and the state required for the update to occur.

2. Notification. The update manager notifies each affected component that an update is available, asking it to converge to a specific state as required by the update. The state could be a generic state, for example *no pending activity*, or a component-specific state, for example *CD-ROM motor off* or *no I/O reads in progress*. The definition of the set of generic states is domain-specific, since each system component must support all of them. Generic states are useful to provide a common framework to specify update constraints at a coarse level of granularity. For greater flexibility and finer control over the update process, each component should also support a set of component-specific states.

3. Preparation. Each component keeps processing work but starts converging to the required state. Components are specifically designed to reach a given state in bounded time. To ensure convergence, a component is allowed to queue new work, with the exception of requests originated from other affected components that must complete to reach the required state globally. When ready, every component saves its state in a safe place and replies back to the update manager, committing itself to maintain the state if not told otherwise. The update manager can abort the update process at any time, for example when a predetermined timeout expires.

4. State checking. When all components have responded, the system has reached the global state desired. To ensure that the entire process is deterministic and reliable, the state of each component is checked for consistency. If any of the components did not meet the required constraints or reached a tainted state due to latent errors, the live update is aborted unless an appropriate recovery mechanism is available.

5. State transfer. The system is now in the state required by the update. The state transfer functions are executed to transform the state of each component into an equivalent state for the new version.

6. Replacement. The new components are now loaded into the running system and initialized with the transformed state. Execution is redirected atomically to the new version and the old components are garbage-collected. A possible optimization is to replace some component incrementally, allowing cross-version execution. This is only safe for components that maintain the exact same behavior in the new version. Developers can mark these components in the live update package.

The proposed model solves the problem of establishing a safe update time structurally, using a deterministic and reliable live update process that is designed to complete in bounded time. The feasibility of an update becomes an implementation problem: a live update is feasible as long as a combination of constraints is available to specify a state of the system that can be transfered to an equivalent state for the new version. To improve update timing, software developers can occasionally reduce the number of constraints on the system at the cost of a higher implementation complexity for the state transfer function.

Admittedly, handwritten state transfer functions can generally represent a fundamental (but unavoidable) threat for every dependable live update solution. Nevertheless, although still required for many categories of updates, writing state transfer functions is made simpler in our model. Each function must only deal with a known and deterministic state of the system, as identified by the update constraints contained in the live update package. The ability to specify the initial state for the update process can greatly reduce (or eliminate) the complexity and the amount of code required for state transfer. In a matter of speaking, the proposed approach shifts large part of the state transfer burden from the update authors to the system itself.

Our model can be considered as a generalization of other update models discussed earlier, but the ability to specify update constraints dynamically can reliably and efficiently support a broader range of updates requiring different levels of atomicity of execution. Our approach can also support updates in an arbitrary state of the system, but is not limited to this simple scenario. Testing is also made simpler and more effective, since changes must only be tested against a deterministic state of the system.

4. Conclusion

In this paper, we have presented a cooperative model for dependable live update that scales efficiently to support a broad range of updates in complex systems. From our analysis, an important aspect emerges: the nature of an update is central in determining the characteristics of the resulting live update process. We argued that existing approaches have largely neglected this important aspect, while focusing more on transparency and backward compatibility.

Our model, in contrast, is based on this observation and aims to build systems that support live update by design. In our approach, the system is receptive to changes and cooperates during the update process to converge to the appropriate state in bounded time before performing the update. Feasibility and safety of a live update are dealt with at design time, while flexibility is guaranteed by the ability to specify update constraints at update time. Better availability can be achieved by trading off the complexity of the state transfer function against the number of constraints imposed on the system.

A legitimate concern is the practical application of the proposed model. We believe, however, that a modular design in which each component has a well-defined interface and an appropriate level of isolation can perfectly fit the live update model we envisioned. Under these assumptions, a component with the required properties could possibly be implemented at different levels of granularity (e.g. module, object, or process). In addition, we believe that our approach can perfectly fit in a typical software development process, also promoting a better system design and encouraging programmers to document changes.

In our ongoing work, we are developing a prototype to apply the ideas described in the paper to operating systems. An operating system is an ideal setting to demonstrate the effectiveness of our model. We are planning to evaluate our prototype system with both small and big updates.

Acknowledgments

This work has been supported by European Research Council under grant ERC Advanced Grant 2008 - R3S3.

References

[1] K. Makris and R. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *Proc. of the USENIX Annual Tech. Conf.*, pp. 397–410, 2009.

[2] J. Buisson and F. Dagnat, "Introspecting continuations in order to update active code," in *Proc. of the First Int'l Workshop on Hot Topics in Software Upgrades*, pp. 1–5, 2008.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proc. of the 18th ACM Symp. on Oper. Systems Prin.*, pp. 73–88, 2001.

[4] P. J. Guo and D. Engler, "Linux kernel developer responses to static analysis bug reports," in *Proc. of the USENIX Annual Tech. Conf.*, pp. 285–292, 2009.

[5] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright, "Timing the application of security patches for optimal uptime," in *Proc. of the 16th USENIX Systems Administration Conf.*, pp. 233–242, 2002.

[6] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Trans. Softw. Eng.*, vol. 22, no. 2, pp. 120–131, 1996.

[7] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: Online patches and updates for security," in *Proc. of the 14th USENIX Security Symp.*, vol. 14, pp. 19–19, 2005.

[8] H. Chen, J. Yu, R. Chen, B. Zang, and P. Yew, "POLUS: A POwerful live updating system," in *Proc. of the 29th Int'l Conf. on Software Engineering*, pp. 271–281, 2007.

[9] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser, "Reboots are for hardware: Challenges and solutions to updating an operating system on the fly," in *Proc. of the USENIX Annual Tech. Conf.*, pp. 1–14, 2007.

[10] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," in *Proc. of the Second ACM SIGOPS/EuroSys European Conf. on Computer Systems*, pp. 327–340, 2007.

[11] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proc. of the Fourth ACM European Conf. on Computer Systems*, pp. 187–198, 2009.

[12] "Ksplice performance record." http://www.ksplice.com/cve-evaluation, 2009.

[13] C. A. N. Soules, D. D. Silva, M. Auslander, G. R. Ganger, and M. Ostrowski, "System support for online reconfiguration," in *Proc. of the USENIX Annual Tech. Conf.*, pp. 141–154, 2003.

[14] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," in *Proc. of the 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 13–24, 2009.

[15] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis, "Contextual effects for version-consistent dynamic software updating and safe concurrent programming," in *Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symp. on Princ. of Progr. Lang.*, pp. 37–49, 2008.

[16] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 72–83, 2006.

[17] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Trans. Softw. Eng.*, vol. 16, no. 11, pp. 1293–1306, 1990.

[18] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 856–868, 2007.

[19] M. Hicks, *Dynamic software updating*. PhD thesis, University of Pennsylvania, 2001.